A Language for Multiple Models of Computation

Dag Björklund Johan Lilius
dag.bjorklund@abo.fi johan.lilius@abo.fi
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, FIN-20520
Turku, Finland

ABSTRACT
We introduce a new kernel language for modeling hardware/software systems, adopting multiple heterogenous models of computation. The language has formal operational semantics, and is well suited for model checking, code synthesis etc. For different blocks of code, different scheduling policies can be applied, to reflect the different interpretations of e.g. parallelism in different models of computation. The user can add his own scheduling policies, to use or explore different models of computation.

1. INTRODUCTION
One important characteristic of modern embedded systems like mobile phones, multimedia terminals, etc. is that their design requires several different description techniques. The radio-frequency part of a mobile phone is designed using analog techniques: the signal processing part can be described using synchronous data-flow, while the protocol stack uses an extended finite state machine based description model.

This heterogeneity poses a challenge to embedded system design methodologies in general, because at the moment it is not possible to obtain comprehensive system models. This is a problem since the increasing integration level of the devices implies that to obtain optimal designs, architectural choices have to be evaluated at an earlier stage than what is possible with current methods. Current approaches force the designer to design the system using one technique (e.g. by writing a functional specification in C++), and to change description technique when making implementation decisions (e.g. by transferring parts to hardware and describing them using a HDL). Because the system description now exists in different languages the comprehensive system model is lost. At the same time, the ability to evaluate and simulate the system as a whole is also lost or becomes very cumbersome.

This problem has been identified and many proposals exist: e.g. Superlog [6], SpecC [7] or SystemC [14] are concrete languages that try to address the problem. On an organizational level several consortia are exploring different System Level Design Languages (e.g. [16, 1]). Most of these proposals suffer from one problem: they are based on some existing language paradigm onto which features from another computational model are glued (e.g. Superlog is based on Verilog, with features like records and pointers from C added on top, while SystemC is class library for C++). These are ad-hoc solutions that appeal to the designer with a background in some programming, or hardware description language.

We believe that to obtain a good System Level Design Language (SLDL) one needs to understand 2 things:

1. what are appropriate description techniques for certain application domains, and
2. how to combine such description techniques into a unified framework.

It is clear that the first point is well understood. This is exemplified by the large amount of programming languages available on the market. It is the second point that we feel is badly understood and it is the long-term goal of our research to obtain such a unified framework.

Central to our work is the notion of a model of computation. A model of computation is a domain specific often-intuitive understanding of how the computations in that domain are done: it encompasses the designer's notion of physical processes, or as Edward A. Lee [11] puts it, the "laws of physics" that govern component interactions. Many different computational models exist: Hardware is often seen as having a *synchronous* model of computation in the sense that everything is governed by a global clock, while software has an *asynchronous* model of computation. [1]. A system that is described using several models of computation is called *heterogenous*.

We are interested in understanding what the combination of models of computation means. Specifically we are interested in understanding the combination of models of computation from an operational perspective. Figure 1 shows an example of a system modeled in two different models of computation: One of the states in a state machine is refined by a Synchronous dataflow (SDF) graph [10]. The intended meaning of this diagram should be: "If event e1 arrives on q1 while in state `wait`, move to state `process` and start processing events using the algorithm in the diagram". However several questions need to be answered before this description can be implemented. For example: what happens if a second `e1` arrives while the system is in state `process`?

[1]We acknowledge that this is a very simplistic point of view.

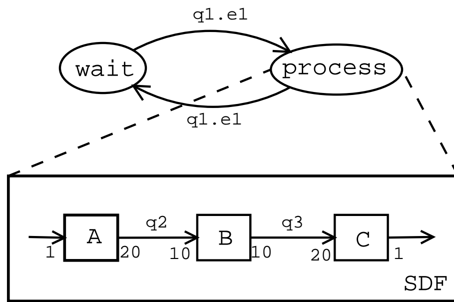

Figure 1: A state machine, with one state refined by an SDF graph

In practice one does not program in a model of computation but in a programming language and we have therefore taken a slightly broader definition and view a model of computation as consisting of both a language and a corresponding semantics. The goal of our research can now be stated as twofold:

1. the development of an unified operational mathematical model of models of computation, and

2. the development of a a textual language, in which it will be possible to program theses kinds of models using the same syntactic elements to represent entities in the different models of computation.

The second goal is motivated by the fact that many of the languages we have looked at (e.g. UML state machines [13], ESTEREL [4] and Harel's Statecharts [9]), use the same syntactic concepts but with different semantics. What we would like to do is pinpoint the semantic differences to certain syntactic concepts. For example the notion of parallelism exists in all three languages above, but there is certainly a difference in the semantics of parallelism between UML state machines and Esterel. On the other hand all languages also have a notion of interrupt (the trap-construct in Esterel and hierarchical transitions in both variants of Statecharts) that seems to have very similar semantics.

The semantics of our kernel language is structured in the following way. We have a structured-operational-semantics (SOS) type semantics for each syntactic entity. This semantics specifies what is common to all models of computation. The differences between models of computations have been encapsulated into functions that we call scheduling policies. This technique allows us to explore the semantic differences and commanilities between different concrete languages.

The main contributions of the paper are:
1. an identification of a number of syntactic entities that are common for many different system level description languages together with a corresponding syntax, and
2. an operational semantics that allows us to attach different interpretations to syntactic elements thus enabling the description of systems using different models of computation.

In section 2, we describe the kernel language, some of its concrete syntax and motivate the choice of syntactic entities. In section 3 we briefly outline the operational semantics, and in section 4 the scheduling semantics of the language. Finally in the last sections, we present some examples and give a conclusion.

1.1 Related Work

Some related work in this area exists. The work of Lee et. al. [12] is a comprehensive study of different models of computation. The authors propose a formal classification framework that makes it possible to compare and express differences between models of computation. It is the only work that we know of that tries to formally define what a model of computation means. The framework is denotational and has no operational content. What this means is, that Lee et. al. are able to describe models of computation that we cannot model in our framework. Such models include timed models and partial order based models. The reason for this is that both timed and partial order based models are models that describe *constraints* on possible implementations. Although we can model data-flow in our language, we have to decide on a specific operational semantics for the data-flow. This semantics will be one of several that preserves the partial-ordering between operations described by the data-flow specification.

On the other hand Girault et al. [8] present ideas for combining different models of computation that are graphically modeled. For example they combine SDF graphs with finite state machines. Their idea is similar to ours in that they use state hierarchy to delineate models of computation.

Finally we would like to point out that in [11], Lee independently, proposes an approach that is conceptually essentially the same as ours, i.e. he suggests that a language, or a set of languages, with a given abstract syntax, can be used to model very different things depending on the semantics and the model of computation connected to the syntax.

2. THE KERNEL LANGUAGE

Our language is a small language, originally designed to describe UML statecharts [5]. Its syntax is given in figure 2. Every statement in a program has a unique label, given by either the designer or a precompiler; for the `state` and `trap` statements, the designer is required to give a label. Looking at different languages that are used for system specification, we have identified a number of concepts that are common, but are interpreted differently given a different computational model.

State The basic concept in our language is the notion of a *state*. State is seldom explicit in programming languages like VHDL or ESTEREL but many modeling languages like UML or Harel's Statecharts or Petri nets make state explicit. Through the notion of state it will be possible to connect an *action language* for describing sequential computations on data as in UML or SpecC. A state is represented by a `state - endstate` block in the language. The state blocks can be hierarchical and several can be active at the same time (concurrency).

Interrupts An interrupt is an event of high priority that should be reacted upon immediately, or almost immediately. In our language, a `trap - endtrap` block is used to monitor interrupts. Interrupts correspond to `trap` in ESTEREL and hierarchical transitions going upwards in the state hierarchy in both UML and Harel's Statecharts.

Coroutines Coroutines are independent threads of control that can be suspended and resumed. The first mainstream language to

if exp **then** $l_1 : stat_1$ **else** $l_2 : stat_2$ **end**	conditional
emit $q.e[(exp)]$	emit an event
dequeue q_1	delete an event
par$(l_1, l_2, \ldots, l_n)$	parallel statement
goto$(l_1, l_2, \ldots, l_n)$	state transition
state [policy][declarations]	start of state block
endstate l	end of state block
trap exp **in**	start of trap block
endtrap l **do** $l_1 : stat_1$	end of trap block
suspend l	suspend a state
resume l_1	resume a state
$[statements]$	atomic block

Figure 2: The statements in our language

explicitly include coroutines was Simula. Later coroutines have been replaced by threads and processes as abstraction mechanisms in programming languages. However in modeling languages coroutines still play a crucial role, e.g. history states in both UML and Harel's' Statecharts label the thread of control in a state as a coroutine, because the state is suspended when a hierarchical transition takes the control out of the state.

Concurrency Concurrency can mean many things. In our language, concurrency is indicated using the `par` statement, where the arguments are the labels of, usually `state` statements, which should be run in parallel. However, the parallelism is interpreted differently depending on the execution policy in effect in the current scope (see below for a explanation of the execution policy).

Atomicity A novelty in our language is that we make atomicity explicit. Atomicity defines what the smallest observable state change is. At the one extreme in traditional programming languages atomicity is not a part of the language itself, but is loosely defined by multiprogramming concepts like semaphores and monitors. The programmer is thus free to define the level of atomicity as he pleases, but this often leads to programming errors as the atomicity does not necessarily follow the scoping rules of the programming language. At the other extreme, in synchronous languages like Esterel, atomicity encompasses the whole program, so that the internal workings of the program are not observable. This is sometimes problematic because the programmer is forced to think of the whole system as a monolothic state machine. In [3] the problem is addressed by *desynchronizing* the large state machine. In the middle-field between these extremes other proposals exist, e.g. the GALS (Globally Asynchronous, Locally Synchronous) semantics proposed in POLIS [2]. Here atomicity is confined to single state machines, while communication between state machines can be observed. In our approach we have introduced atomicity as an explicit syntactic entity, the atomic brackets `[ ]`. It abides to the normal rules of scoping and is thus less general than the first approach mentioned above. But using this approach we can model its interaction with the other constructs to obtain exactly the level of atomicity needed.

Communication policy The communication policy states how different modules of the system communicate with each other. Several alternatives exist. For the moment we have taken a rather simple approach which allows us to still model many more complex approaches.

We call the main communication media in our language, *channels*. A channel can e.g. represent the global event queue in a UML statechart, a link in an SDF graph etc. In state diagrams, an event is an occurrence that may trigger a state transition. In UML statecharts, there is an implicit global event queue; whereas, in our language several channels can be declared and the scope of a channel declaration is the `state` block. The notation in our language for checking for the presence of an event on a queue is `q1.e1`, where `q1` is the queue and `e1` is an event. The events can also have values of some built in type, or a type defined in the target language.

Data Data handling is not our primary concern at the moment, as we are more interested in control-dominated programming; however, the language has a few primitive types like integers and floats. Complex types, as well as functions and procedures are only declared in the language, while their implementation is deferred to the target language. This is the same approach as in ESTEREL.

The main elements in a program in our language are the `state` blocks. We could also call them e.g. actors or agents (the term actor-oriented design is introduced in [11]), since depending on the model of computation, a state block can represent e.g. a state in a statechart or a node in an SDF graph etc.

3. OPERATIONAL SEMANTICS

The labels of the statements act as values for program counters. The state of a program is represented by a tuple $< \alpha, suspend, q >$ where:

- $\alpha \subseteq \Sigma$ is the set of active labels, Σ is the set of all labels.
- $suspend\ \Sigma \to \Sigma$ is a partial function. If $suspend(l) = l'$ then l is a suspended state and l' is a substate of l that was active when l was suspended.
- q is the set of event queues.

The active set represents the active threads of execution, and the labels represent the current values of the program counters. Some of the threads may be suspended. The operational semantics is given by a set of rules that determine actions of the form:

$$\frac{premiss}{< \alpha, suspend, q > \xrightarrow{statement} < \alpha', suspend', q' >},$$

e.g. if the premiss holds the system can change from state $< \alpha, suspend, q >$ to state $< \alpha', suspend', q' >$ by executing *statement*. We sometimes use the abbreviation σ for the state tuple. A subset of the rules is given figure 3 [2] . The rules determine how the statements update the state of the machine.

An *execution engine* picks labels from the set of active states and then executes the rule corresponding to the statement at the label. An *execution policy* decides on the order in which labels are picked from the active list. We shall describe the issue of policies more in depth below.

Before we can look more in detail at the operational rules we need to formalize the structure of a program. A program is defined by

[2] We use the domain/range restriction operators $\lhd$ / $\rhd$ and the domain/range subtraction $⩤$ / $⩥$ operators as defined in Z [15] to operate on the relations: Let S be a set and R a relation, then $R \lhd S$ is a restriction of R, where every element in the domain of R is a member of the set S. The domain/range subtraction operators are similar, except they remove the elements in the set from the relation.

the tuple $\langle \Sigma, \uparrow, \succ, \mathcal{L} \rangle$ where: Σ is the set of labels. $\uparrow \subset \Sigma \times \Sigma$ is the *parent* relation. The parent relation organizes the labels in Σ hierarchically in a tree. We can take the closure $\uparrow^*$ of the parent relation $\uparrow$. Let l_1 and l_2 be labels, then $l_1 \uparrow^* l_2$ means that l_1 is an *ancestor* of l_2 or in other words: l_2 is inside the scope of l_1. $\succ \subset \Sigma \times \Sigma$ is the *immediate successor* relation. If $l_1, l_2 \in \Sigma$ and $l_1 \succ l_2$ then l_2 succeeds l_1 in the program. $\mathcal{L} : \Sigma \rightarrow statement$ is a total function that maps each label to its statement.

There are in total 18 rules that define the semantics of the language. In this paper we only show a few due to the space limitation.

Parallel statement A `par` statement can act when its label is active; it adds all of its arguments to the active set. Note that the par statement just creates new threads. It is the job of the policy to decide how to schedule the threads.

Goto The `goto` statement is used to do state transitions. The only way to escape a state block, is by using the `goto` statement. It can take many labels as parameters, which allows it to be used to model fork transitions. It is enabled when its label is active; it removes the subtree containing the source and target labels from the active set, and activates the target labels (belonging to state statements) along with any labels labeling `trap` statements that are ancestors of the target labels.

State A `state` statement can act when its label is active. It will remove its label from the active set and add its successor to the active set. It marks the beginning of a state block.

Endstate An endstate statement can act when its label is active. It removes its label from the active set, and reactivates the label of the corresponding state statement. After this, the scheduler can execute the state block again.

Sequencing When a statement succeeded by a '`;`' is executed, the label of the statement that succeeds this statement in the code becomes active.

4. SCHEDULING SEMANTICS

We have presented the concrete syntax, and the execution semantics of the language; however, there is still a freedom in how the active labels are scheduled. We introduce an *execution policy* for different models of computation that, connected to a `state`, will schedule the substates according to the model of computation. A program is executed by repeatedly running the policy of the topmost state in the hierarchy. We well define *RUN* functions implementing the different execution policies. Each state in a program has a *RUN* function that executes a policy. The topmost policy will then schedule the states down in the hierarchy, which can have different policies assigned to them, thus having a different model of computation. The entity that calls the *RUN* function of the top-level state can be thought of as a global clock in the system.

To define the *RUN* functions we need some helper functions. A label that belongs to a simple statement is *enabled* iff the premiss of the rule that corresponds to the statement holds; if a label belongs to a state statement, it is enabled iff it has descendants that are enabled. A label can also become blocked by the scheduler, and is then not enabled. The boolean function $IS_ENABLED(l, \sigma, \beta)$ returns true if label l is enabled when the program is in state σ and the labels in set β are blocked: ($premiss(l, \sigma)$ returns true if the premiss of the rule for l holds in state σ)

IS_ENABLED(l, σ, β)
1 **if** $\mathcal{L}[l] \neq state$
2   **return** $premiss(l, \sigma) \wedge l \notin \beta$
3 **else**
4   **return** $ENABLED(l, \sigma, \beta) \neq \emptyset$

The $ENABLED(l, \sigma, \beta)$ function returns the set of enabled labels that are descendants of l:

ENABLED(l, σ, β)
1 **return** $\{l_i : label \mid IS_ENABLED(l_i, \sigma, \beta) \wedge l \uparrow^* l_i\}$

Now we can define the *RUN* functions for the scheduling policies. The functions take the label of the state block l, the current state of the program σ and the blocked set β as parameters and return the next state of the program.

UML Statechart Semantics (run-to-completion) The semantics of UML statecharts is based on the run-to-completion (RTC) step. The rtc policy executes the UML statechart semantics. A state executing the rtc policy runs until no more runnable labels exists in its scope. It first runs all the enabled statements, and then checks if there are any more enabled labels left, adding any labels that are not ancestors of the original enabled labels to the blocked set (line 5). If there are enabled non-blocked labels, it runs again. By this blocking, we ensure that when a state transition is taken during a rtc step, the newly activated state will not be executed during that same step.

RUN(l, σ, β)
1 $\rho \leftarrow ENABLED(l, \sigma, \beta)$
2 **for each** l_i **in** ρ **do**
3     $\sigma \leftarrow RUN(\sigma, l_i, \emptyset)$
4 $\rho' \leftarrow ENABLED(\sigma', l, \emptyset)$
5 $\beta' \leftarrow \{l_j \in \rho' \mid \mathcal{L}[l_j] = state \wedge l_j \notin (\ ran(\rho \lhd \uparrow^*\ )\}$
6 **if** $\rho' - \beta' \neq \emptyset$ **then**
7   **return** $RUN(l, \sigma', \beta')$
8 **return** σ

Interleaving Semantics The policy for the interleaving semantics picks a label from the enabled set nondeterministically under the fairness assumption (The $:\in$ in line 2 denotes nondeterministic assignment).

RUN(l, σ, β)
1 $\rho \leftarrow ENABLED(l, \sigma, \beta)$
2 $l_i :\in \rho$
3 **return** $RUN(l_i, \sigma, \emptyset)$

Synchronous Dataflow Semantics The SDF semantics schedules the substates according to a static schedule S. The schedule is an circular array of labels, assumed to belong to `state` statements. The SDF policy runs one of the substates in the schedule per invocation, and has a variable to keep track of where it is in the schedule.

RUN(l, σ, β)
1 $\sigma' \leftarrow RUN(S.current(), \sigma, \emptyset)$
2 $S.advance()$
3 **return** σ'

Parallel statements

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = par(l_1, l_2, \cdots, l_n)}{<\alpha, q> \xrightarrow{l:par(l_1, l_2, \ldots, l_n)} <\alpha - \{l\} \bigcup_{i=1}^{n} \{l_i\}, q>}$$

Goto

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = goto(l_1, l_2, \cdots, l_n)}{<\alpha> \xrightarrow{goto(l_1, l_2, \cdots, l_n)} <\alpha - ran(\{l_{min}\} \lhd \uparrow^{+}) \bigcup_{i=1}^{n} trapanc(l_i) \cup \{l_1, \ldots, l_n\}>}$$

State

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = state \wedge l \succ l_1}{<\alpha, suspend, q> \xrightarrow{l:state} <\alpha - \{l\} \cup \{l_1\}, suspend, q>}$$

Endstate

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = endstate\ l'}{<\alpha, q> \xrightarrow{endstate\ l'} <\alpha - \{l\} \cup \{l'\}, q>}$$

Sequence

$$\frac{<\alpha, q> \xrightarrow{l_1:stat_1} <\alpha - \{l_1\}, q'> \wedge\ l_1 \succ l_2}{<\alpha, q> \xrightarrow{l_1:stat_1;} <\alpha - \{l_1\} \cup \{l_2\}, q'>}$$

Figure 3: A subset of the operational rules of the language

5. EXAMPLES

We will now demonstrate the use of the language by two examples.

A State Machine with SDF subsystem

The model shown in section 1 can be written in our language as depicted in Figure 4. The whole system is represented by the `S` state. The `wait` and `process` states represent the two states in the state machine, the `rtc` policy connected to the `S` state (line 4) schedules these states according to the UML statechart semantics. The transition from `wait` to `process` is triggered by the `goto` statement in line 7, guarded by the `if` statement that checks for the presence of event `start` on channel `q1`. The `process` state schedules its substates according to the sdf policy with the given static schedule (line 13). The `B` state is executed twice for each invocation of the `A` and `C` states since it consumes half the amount of tokens that `A` produces etc. The channels connecting the SDF nodes can be given capacities as in line 12. The transition back to the `wait` state is achieved by the `goto` statement in line 26, which is triggered by the `trap` block which encloses the whole SDF subsystem.

```
1   S: state
2     channel q1;
3     event start, stop;
4     policy rtc;
5     wait: state
6       if q1.start then
7         goto( process )
8       endif;
9     endstate wait
10    l1: trap q1.stop in
11      process: state
12        local channel q2(20), q3(40);
13        policy sdf(A,B,B,C);
14        par( A, B, C )
15        A: state
16          # code block for A
17        endstate A
18        B: state
19          # code block for B
20        endstate B
21        C: state
22          # code block for C
23        endstate C
24      endstate process
25    endtrap l1 do
26      goto( wait )
27  endstate S
```

Figure 4: Example Program 1

A UML Object Collaboration Diagram

As another example, we can model a collaboration of two active objects `A` and `B` that have behavior modeled by state machines, as depicted in figure 6. We use the interleaving policy to schedule the two objects modeled by states (line 5 and 13). The `par` statement in line 4 will activate the two states, and the scheduler will repeatedly execute one step for one of them. A step in this case will be a complete execution of the whole state block i.e. state machine of the object, since they are internally scheduled by rtc policies.

```
1   collab: state
2     policy interleaving;
3     channel q1;
4     par( A, B )
5     A: state
6       policy rtc;
7       a: state
8         l1: if q1.e1 then goto( b ) endif;
9       endstate a
10      b: state
11        l2: if q1.e1 then goto( a ) endif;
11      endstate b
12    endstate A
13    B: state
14      channel q2;
15      policy rtc;
16      c: state
17        if q2.e2 then emit( q1.e1 ) endif;
18      endstate c
19    endstate B
20  endstate collab
```

Figure 5: Example Program 2

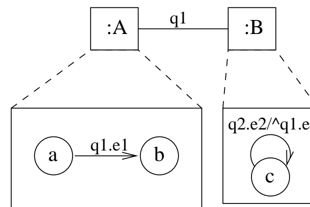

Figure 6: A collaboration diagram with two active objects A and B, with behaviour described by state machines

In the previous example, we stated that the rtc scheduler runs its state block to completion, when it is called by the higher level interleaving scheduler. This will happen in one atomic step. The observable state transitions in the state machine of object `A` is shown by the automaton figure 7 (a). If we were to replace the rtc policy

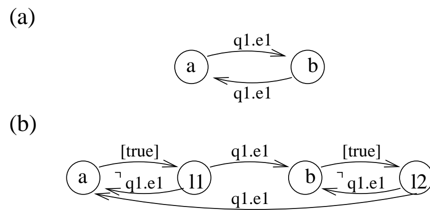

Figure 7: Automaton describing the level of atomicity, when using (a) rtc and (b)interleaving policies for the state machine of object A in figure 6

with an interleaving policy, we could observe state changes something like in the automaton in 7 (b).

In this case we could also achieve the rtc semantics using the interleaving policy, by encapsulating the state machine states with atomic brackets `[ ]`. The scheduler would then again only run one statement per step, but since an atomic region is executed like a single statement, we would again get the same behavior as in 7(a).

6. CONCLUSIONS AND FURTHER WORK

We have presented a kernel language for describing systems with several underlying models of computation. The novelty of our approach lies in the way we have separated the semantics of the model of computation from other concepts like interrupts and communication.

We currently have a C code generator based on an execution engine, i.e. we have a library containing implementations of the operational rules as well as the scheduling policies, againts which the generated code is linked. In the future we wish to adopt a more optimal method, where we translate the code into finite state machines, which we optimize. This method is demonstrated for a subset of the language in [5] and can easily be used for generating assembly, HDLs etc.

In [11] Lee, enumerates some models of computation, e.g. dataflow, publisher/subscriber etc. We already discussed data-flow in the introduction. At the moment the communication concepts in the language are undeveloped, but a publisher/subscriber like model of computation can easily be modeled using the existing communication mechanisms.

One of the problems in the present version of the language is that the policies are expressed in an ad-hoc notation. We would like to be able to express also the policies in our kernel language. This will require the development of a metamodel for our language.

7. ACKNOWLEDGMENTS

We wish to thank the four anonymous reviewers of a previous version of this work.

8. REFERENCES

[1] The Accellera Consrotium web site. *Internet: http://www.accellera.org*.

[2] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems*. Kluwer Academic Publishers, 1997.

[3] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *LNCS*. Springer Verlag, 1999.

[4] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[5] D. Björklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. In *pUML Workshop at UML2001*, october 2001.

[6] P. L. Flake and S. J. Davidmann. Superlog, a unified design language for system-on-chip. In *Proceedings on the 2000 conference on Asia and South Pacific design automation*, pages 583 – 586. ACM Press, 2000.

[7] Gajski, Zhu, Dömer, Gerstlauer, and Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[8] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6), june 1999.

[9] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Tran. of Software Engineering and Methodology*, 5(4) Oct 1996.

[10] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9), September 1987.

[11] E. A. Lee. Embedded software. *Advances in Computers (to appear)*, 56, 2002.

[12] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Tran. on CAD*, 17(12), 1997.

[13] Object Management Group. *Unified Modeling Language Specification 1.3*, chapter 3.74. OMG, june 1999.

[14] The Open SystemC Initiative web site. *Internet: http://www.systemc.org*.

[15] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[16] The System Level Design Consortium web site. *Internet: http://www.sldl.org*.